Join Ordering for Constraint Handling Rules
Putting Theory into Practice

Peter Van Weert*

Department of Computer Science, K.U.Leuven, Belgium
Peter.VanWeert@cs.kuleuven.be

Abstract. Join ordering is the NP-complete problem of finding the optimal order in which the different conjuncts of multi-headed rules are joined. Join orders are the single most important determinants for the runtime complexity of CHR programs. Nevertheless, all current systems use ad-hoc join ordering heuristics, often using greedy, very error-prone algorithms. As a first step, Leslie De Koninck and Jon Sneyers therefore worked out a more realistic, flexible formal cost model. In this work-in-progress paper, we show how we created a first practical implementation of static join ordering based on their theoretical model.

1 Introduction

Constraint Handling Rules (CHR) [6] is an elegant, very high-level programming language based on multi-headed guarded rules. Originally designed for the declarative specification of constraint solvers, CHR is increasingly used as a general purpose programming language, in a wide range of applications [15]. A considerable amount of research is devoted to the optimizing compilation and execution of CHR programs [5,11,19,22], and efficient, state-of-the-art implementations exist for Prolog [11,12], HAL [5,8], Java [21], Haskell, and C [23].

The most critical part of any rule-based system is the search for matching partner constraints to form applicable rule instances, given an active—typically just added—CHR constraint. To prune this search space many techniques are used, including loop-invariant code motion (e.g. testing guards as soon as possible) and constraint store indexing (cf. Example 1). Their applicability and effectiveness is almost always completely determined by the order in which the partner constraints are joined.

Example 1. The following rule occurs in the CHR-based RAM simulator of [14]:

```
pc(L), mem(A,X) \ prog(L,ADD,B,A), mem(B,Y) ⇔ mem(A,X+Y), pc(L+1).
```

It implements the ADD instruction of the simulated RAM machine. The different CHR constraints model the RAM machine's program counter (`pc/1`), memory (`mem/2`), and program (`prog/4`).

Suppose a new `pc(L)` constraint is added. To determine whether the above rule is applicable, a naive implementation would match the different conjuncts of the head in textual order. This entails enumerating all `mem/2` constraints, for each of them checking whether a suitable `prog/4` constraint is in the store. Even if e.g. a hash- or array-based index is used to check for matching `prog/4` constraints in $\mathcal{O}(1)$ time, this process remains linear in the size of the RAM machine's memory.

With the correct join order, the runtime would first look up a matching `prog/4` constraint using the `L` value known from the active `pc(L)` constraint, and only then retrieve the two `mem/2` constraints. This way, given proper indexing, the evaluation of the rule occurs in optimal constant time, instead of the naive linear time.

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).

Finding optimal join orders is thus quintessential for the optimal time complexity of most CHR programs. The join ordering problem, however, is NP-complete [9], and may moreover depend on dynamic properties such as the size of the constraint store, the selectivity of guards, etc. Current state-of-the-art optimizing compilers therefore use ad-hoc heuristics to determine join order, mostly based on those proposed in [5,8]. They moreover mostly use ad-hoc algorithms to minimize the estimated cost.

As a first step towards more effective join ordering for CHR, [2,3,13] therefore worked out a reasonable, more realistic cost formula, and discussed in detail how to heuristically approximate it either statically or dynamically. They moreover proposed several techniques adapted from database literature [10,16,18] for implementing join ordering based on their model. Unfortunately, their descriptions are very sketchy and contain errors, which we will point out and correct in this paper.

This work-in-progress paper thus represents a necessary second step, transforming the theoretical principles of [2,3,13] into correct, practical join ordering algorithms. We first explain three join orderers we implemented for JCHR*2* (Sections 3–4), an upcoming new CHR system for Java [21]. Next, Section 5 briefly lists some considerations for implementing the more efficient 'KBZ' algorithm proposed in [2,3,13], and Section 6 compares with related work. The next step, part of future work (Section 8), will involve validating, fine-tuning, and improving the cost formula, our heuristics and our algorithms based on more extensive experimentation.

2 Problem Statement

In this section, we very briefly and informally reconstruct the cost formula derived in [2,3,13]. More rigorous definitions can be found in these references.

Slightly simplified[1] and reordered, and under a number of reasonable assumptions (e.g. only $\mathcal{O}(1)$ equality indexes are used, and all remaining—so-called *a posteriori*—guards are also evaluated in constant time) and restrictions (e.g. nested loop joins only), the cost of matching n heads according to a given join order Θ is:

$$C_\Theta^{[1..n]} = \sum_{j=1}^{n} |\mathcal{J}_\Theta^{j-1}| \cdot \mu^\Theta(j) = \sum_{j=1}^{n} \prod_{k=1}^{j-1} \left(\mu^\Theta(k) \cdot \sigma_\star^\Theta(k)\right) \cdot \mu^\Theta(j) \tag{1}$$

with (all defined assuming partners are joined in the order determined by Θ):

- $\left|\mathcal{J}_\Theta^k\right| = \left|\mathcal{J}_\Theta^{k-1}\right| \cdot \mu^\Theta(k) \cdot \sigma_\star^\Theta(k)$ the size of a partial join: the number of CHR constraint tuples that match the first k heads; we call these *k-tuples*;
- $\mu^\Theta(k)$ the (average) *multiplicity*: the average number of constraints that satisfy the k'th partner's *a-priori guards*—the guards that are tested a priori using a constraint index—per $(k-1)$-tuple for which at least one k-tuple exists; and
- $\sigma_\star^\Theta(k)$ the (average) *selectivity*: the average percentage of these k-tuples that satisfy the k'th partner's *a-posteriori guards*—the remaining guards.

Our join ordering problem is thus finding a join order Θ that minimizes the cost formula (1). We refer to [2,3,13] on detailed discussions on how to (statically) estimate the $\mu^\Theta(k)$ and $\sigma_\star^\Theta(k)$ factors.

3 Exhaustive Algorithms

3.1 Branch and bound join ordering

The most straightforward join ordering algorithm exhaustively enumerates all $n!$ possible join orderings. It can be viewed as traversing a tree with the empty join at the root, and complete join orderings at the leaves, in a depth-first, left-to-right

[1] Concretely, for simplicity, we ignore the $\sigma_{\text{eq}}^\Theta(k)$ factor of the actual cost formula. This is reasonable, since any static estimate assumes $\sigma_{\text{eq}}^\Theta(k) = 1$ (cf. [2,3,13] for details).

order. For this, it is more convenient to rewrite (1) as follows:

$$C_{\Theta}^{[1..n]} = \sum_{j=1}^{n} \prod_{k=1}^{j} \left(\sigma_{\star}^{\Theta}(k-1) \cdot \mu^{\Theta}(k)\right)$$

It then becomes apparent that this sum can be efficiently computed incrementally:

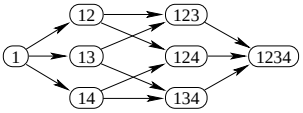
$$\begin{cases} C_{\Theta}^{[1..0]} = 0 \\ C_{\Theta}^{[1..i]} = C_{\Theta}^{[1..i-1]} + \vartheta^{\Theta}(i) \end{cases} \text{ with } \begin{cases} \vartheta^{\Theta}(1) &= \mu^{\Theta}(1) \\ \vartheta^{\Theta}(i+1) &= \vartheta^{\Theta}(i) \cdot \sigma_{\star}^{\Theta}(i) \cdot \mu^{\Theta}(i+1) \end{cases}$$

To slightly optimize this algorithm, we use the standard branch and bound technique to filter the search space. That is, we keep the currently minimal cost of leaf (a complete join) C_{min}, and stop traversing the tree as soon as $C_{\Theta}^{[1..i]} \geq C_{min}$.

As the time complexity of this join orderer clearly is $\mathcal{O}(n!)$ (and the space complexity $\mathcal{O}(n)$), it is only useful for rules with very few heads.

3.2 A⋆ join ordering

A second, more efficient exhaustive join ordering algorithm is based on A⋆ (we assume the reader is familiar with this standard algorithm [7]). The basic algorithm maintains a pool of partial joins $\mathcal{J}_{\Theta}^{k}$ (initially a single, empty join $\mathcal{J}_{\Theta}^{0}$). In each iteration, the most promising partial join $\mathcal{J}_{\Theta}^{k}$ is heuristically selected, and gives rise to $n-k$ new partial joins, one for each remaining join partner. Clearly, this way the worst-case time and space complexity remains $\mathcal{O}(n!)$.



However, suppose the two partial joins (123) and (132) have the same cost, then expanding them both is pointless. We therefore use a standard A⋆ optimization where a *closed set* of already expanded joins is kept, where we treat joins such as (123) and (132) as identical. Essentially, this reduces to problem to finding a shortest path in a DAG such as illustrated to the right (for $n = 3$). The worst-case time and space complexities are thus reduced to $\mathcal{O}(n \cdot 2^n)$ and $\mathcal{O}(2^n)$ respectively.

For a given set of remaining, not-yet-joined partners, the A⋆ algorithm requires a heuristical lower bound on the estimated cost of computing the remainder of the join. This heuristic must be *admissible*, that is, it may never exceed the actual remaining cost estimate given by the cost formula (1). Suppose a partial join has length k. Let Θ be any join order starting with the k already fixed partners. Then the actual cost (1) of joining the remaining partners X in that order is of the form:

$$C(X) = |\mathcal{J}_{\Theta}^{k}| \cdot \sum_{i=1}^{n-k} \left(\left(\prod_{j=1}^{i-1} \mu^{\Theta}(k+j) \cdot \sigma_{\star}^{\Theta}(k+j) \right) \cdot \mu^{\Theta}(k+i) \right) \tag{2}$$

Because $|\mathcal{J}_{\Theta}^{n}|$ depends only on the already fixed partial join, the problem is reduced to finding a heuristic H that is an efficiently computable tight lower bound on the remaining sum. The following two concepts will be crucial for this:

- The *minimal multiplicity* μ^{min} of a head conjunct is heuristically estimated as the expected number of constraints that satisfy the (implicit) a-priori equality guards on the conjunct's arguments, assuming all shared variables are given (or in other words: assuming it is looked up as the last partner in the join order, using optimal equality indexing).
- The *maximal (a-posteriori) selectivity* $\sigma_{\star}^{max}$ is heuristically estimated as the expected probability that the a-posteriori guards hold for a given constraint matching the a-priori guards, again assuming all these guards can be tested. The *maximal* selectivity is actually the *minimal* probability of entailment.
- We further define $\gamma^{min} = \mu^{min} \cdot \sigma_{\star}^{max}$ for each partner, intuitively the *minimal cardinality* of the set of constraints matching a head conjunct.

For each partner, these estimates only have to be computed once (cf. [2,3,13] for a detailed description on estimating multiplicities and selectivities).

Original heuristic Let C_X be the sequence of γ^{min} values of the partners in X, sorted from small to large, and M_X^0 and S_X^0 the sequences of μ^{min} and $\sigma_\star^{max}$ values of the corresponding heads, that is: $\forall i : C_X[i] = M_X^0[i] \cdot S_X^0[i]$. To compute the C_X, M_X^0, and S_X^0 sequences, it suffices to sort all conjuncts of a given head once.

Using this notation, the heuristic proposed by [2] and [13] is given by:

$$H_0(X) = \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} C_X[j] \Big) \cdot M_X^0[i] \tag{3}$$

Unfortunately, this heuristic is inadmissible. The premise of this heuristic is that, by sorting the γ^{min} values, the sum in (3) is minimized. To show that this premise does not hold, suppose we swap the elements α and β of sequences C, M_0 and S_0 ($1 \leq \alpha < \beta \leq n-k$). The sum then becomes:

$$H_0'(X) = \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} C_X'[j] \Big) \cdot M_X'^0[i]$$

Clearly, the terms for $i < \alpha$ and $i > \beta$ remain unchanged after swapping, and

$$\begin{aligned} H_0(X) - H_0'(X) = \Big(\prod_{j=1}^{\alpha-1} C_X[j] \Big) \cdot \Big(& M_X^0[\alpha] - M_X^0[\beta] \\ & + \big(C_X[\alpha] - C_X[\beta]\big) \cdot \big(M_X^0[\alpha+1] + \ldots\big) \\ & + \big(S_X^0[\alpha] - S_X^0[\beta]\big) \cdot M_X^0[\alpha] \cdot M_X^0[\beta] \cdot \prod_{k=\alpha+1}^{\beta-1} C_X[k] \Big) \end{aligned}$$

If the heuristics' premise were correct, then $H_0(X) \leq H_0'(X)$. But then not only must $C_X[\alpha] \leq C_X[\beta]$, but also $M_X^0[\alpha] \leq M_X^0[\beta]$ and $S_X^0[\alpha] \leq S_X^0[\beta]$. In general, however, sorting the products of M^0 and S^0 does not guarantee that the sequences themselves are sorted. A counter-example is easily obtained by $C = [1, 3]$, $M^0 = [2, 15]$ and $S^0 = [0.5, 0.2]$, where the latter is unsorted.

Correct heuristics A first correct underestimate is derived as follows. Observe that the ith term in the sum of the actual cost (2) is given by a product of i $\sigma_\star^\Theta$ and $i+1$ μ^Θ values. A correct lower bound for the ith term is thus the product of the i smallest $\sigma_\star^{max}$, and the $i+1$ smallest μ^{min} values. First, we therefore sort both the $\sigma_\star^{max}$ and the μ^{min} values of all occurrences in X in two sequences S_X and M_X. Again, in practice, two global S and M lists are computed, from which S_X and M_X are readily derived. The heuristic H_1 is given by:

$$H_1(X) = \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} M_X[j] \cdot S_X[j] \Big) \cdot M_X[i]$$

This heuristic only coincides with H_0 if the sequences M_X^0 and S_X^0 happen to be sorted, which as shown earlier is not always the case.

In H_1, we observe that $M[i]$ and $S[i]$ generally do not originate from the same occurrence, while in the actual cost, the $\mu^\Theta(i) \cdot \sigma_\star^\Theta(i)$ factors do belong to the same occurrence. An alternative underestimate is thus based on a sequence C_X defined as before, and the smallest minimal multiplicity of all occurrences in X, i.e. $M_X[1]$:

$$H_2(X) = \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} C_X[j] \Big) \cdot M_X[1]$$

Again, each term clearly underestimates the corresponding term in the actual cost. The difference with H_0 is that instead of multiplying with $M_X^0(i)$, each term is multiplied with $M_X[1]$, a trivially safe (yet possibly very poor) underestimate.

When comparing H_1 and H_2, there is no clear winner. Obviously

$$H_1(X) = M_X[1] \cdot \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] \Big)$$

and therefore

$$H_1(X) - H_2(X) = M_X[1] \cdot \sum_{i=1}^{n-k} \Big(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] - \prod_{j=1}^{i-1} C_X[j] \Big)$$

Neither of the above heuristics is thus superior in itself. The heuristic currently used by JCHR*2* is therefore simply

$$H_3(X) = \max\big(H_1(X), H_2(X)\big)$$

It provides fairly tight lower bounds, while still remaining admissible and efficiently computable. We need only to compute three sorted sequences M, S, and C containing the values for all join partners of a head once. Using these sequences, computing the heuristic H_3 has a reasonable runtime cost linear in the number of remaining partners.

4 Randomized Algorithms

While our A* join orderer scales reasonably well, it remains an exponential algorithm. In fact, as join ordering is NP complete [9], any exhaustive algorithm is bound to be infeasible in general. For really large heads (currently $n > 10$ in JCHR*2*), we must therefore fall back to randomized algorithms that compute reasonable—though not necessarily optimal—join orders in reasonable time.

Our current implementation uses local search algorithms inspired by the 'iterative improvement' algorithm of [17]. This algorithm is essentially a random-restart hill climbing algorithm, but we extended it to a random-restart beam search algorithm. Starting from some initial join ordering (chosen either randomly, or using some greedy join ordering algorithm), this join order is incrementally improved, by randomly generating small changes (e.g. swapping two partners; cf. [17]), and updating the current order each time such a change results in a cost improvement. In the beam search variant, a fixed set of the b best join orders is kept instead of just the one. Once a threshold of subsequent unsuccessful local changes is met, the algorithm repeats the same process with a different (pseudo-random) initial join ordering. The algorithm ends, once some stop criterium is met (currently either a fixed number of (unsuccessful) restarts, or some timeout polynomial in n).

Space limitations prohibit a more detailed description. In any case, more experimentation is needed to tune the many parameters of the algorithm (cf. Section 8).

5 On the KBZ Algorithm

In [2,3,13], an $\mathcal{O}(n \log n)$ algorithm is presented for the join ordering of a common, specific type of rule heads (those with acyclic join graphs to precise; cf. [2,3,13]). It is currently not yet implemented in JCHR*2*. Still, our preliminary analysis already revealed the following two issues, relevant to anyone who wants to implement it:

1. In [2,3,13], wrongfully call their algorithm a KBZ algorithm, and accredit it to [10]. The algorithm they actually describe is the IK algorithm, which was first applied to join ordering by [9]. The real KBZ algorithm of [10] further improves on the IK algorithm. When applied to our problem, it computes the join order for all n active constraints of a given head in $\mathcal{O}(n^2)$ time instead of $\mathcal{O}(n^2 \log n)$.

2. We believe the version of the IK algorithm described by [10], and subsequently copied by [2,3,13], is not correct. It uses a step where chains of nodes are merged, but the problem is that these chains may not be sorted. The normalisation it performs at the root of the merged chains does not resolve this. The true IK algorithm correctly normalises (sorts) both chains before merging [9].

6 Related work

Join ordering received considerable attention in database research [9,10,18], too much to cover here. We refer e.g. to [16] for a good survey on randomized join ordering algorithms.

CHR implementations currently use ad-hoc join ordering heuristics, typically based on those described by [5,8]. We refer to [2,3,13] for a detailed discussion why the cost model underlying these heuristics is flawed. The algorithms used to minimize the estimated cost, moreover, are mostly crude and ineffective. Both HALCHR [5,8] and the initial JCHR system [21] use a linear, greedy algorithm, that often leads to suboptimal results. The K.U.Leuven CHR system uses a naive A*-based algorithm (i.e. without closed set), but only if $n < 6$. For larger multi-headed rules, the partners are simply joined left-to-right. The CCHR [23] system uses a straightforward branch-and-bound optimization algorithm for $n \leq 8$; for larger heads it simply generates 40,000 random join orders and keeps the best. Clearly, given the importance of join ordering, settling for such ad-hoc algorithms cannot be excused.

7 Conclusions

Join ordering is fundamental for the optimal runtime complexity of CHR programs. Nevertheless, both the heuristics (cf. [2,3,13]) and algorithms (cf. Section 6) used by current systems are very ad-hoc. The first issue was addressed by [2,3,13], the latter in this paper. Practice shows that, unlike claims to the contrary in [5,8], CHR programs do frequently contain complex multi-headed rules ([2,3,13] provide examples). The careful design and implementation of adequate join ordering algorithms is therefore indispensable. We outlined how to translate the theoretical model of [2,3,13] into efficient, flexible join ordering algorithms. For JCHR*2*, we implemented three join orderers, based on branch-and-bound, A*, and local search respectively, each used for increasingly larger rule heads. We also listed some considerations on implementing a more efficient poly-time KBZ algorithm (as first proposed in [2,3,13]). The contributions reported in this paper are relevant to anyone who wants to implement join ordering (based on [2,3,13]).

8 Future work

The current combination used by JCHR*2* seems to work well in practice. Still, we only have scratched the surface, and more experimentation is required to determine:

1. whether the assumptions made by the cost function of [2,3,13], and the heuristics used to estimate it, are indeed appropriate;
2. the optimal parameters for the local search algorithm (starting points, local moves, beam size, stopping criteria, etc.). Also, alternative randomised algorithms (genetic algorithms, simulated annealing, etc.) could be considered [16].

Many more issues must be further investigated: first-few answers (cf. [1,2,13]), join strategies besides nested-loop joins, a-priori guards besides equality, etc.

The most important open problem though is that, short of reliable estimates for e.g. cardinalities and selectivities, static join ordering frequently will always remain error-prone. To mend this shortcoming, we proposed annotations that allow the

user to specify cardinalities and selectivities in [20]. While these annotations help, they rely on the programmer to supply sufficient and correct information. The only really efficacious solution is dynamic join ordering (cf. [2,3,13,19]).

References

1. R. J. Bayardo, Jr. and D. P. Miranker. Processing queries for first-few answers. In *CIKM '96: Proc. fifth intl. Conf. Inf. and Knowl. Mgmt.*, pages 45–52. ACM, 1996.
2. L. De Koninck. *Execution Control for Constraint Handling Rules.* PhD thesis, K.U.Leuven, Belgium, Nov. 2008.
3. L. De Koninck and J. Sneyers. Join ordering for Constraint Handling Rules. In Djelloul et al. [4], pages 107–121.
4. K. Djelloul, G. J. Duck, and M. Sulzmann, editors. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules*, Sept. 2007.
5. G. J. Duck. *Compilation of Constraint Handling Rules.* PhD thesis, University of Melbourne, Australia, Dec. 2005.
6. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, (37):28–29, 1972.
8. C. Holzbaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. volume 5(4–5) of *Theory and Practice of Logic Programming*, pages 503–531. Cambridge University Press, July 2005.
9. T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.
10. R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB '86: Proc. 12th Intl. Conf. on Very Large Data Bases*, pages 128–137, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
11. T. Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules.* PhD thesis, K.U.Leuven, Belgium, June 2005.
12. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *CHR '04, Selected Contributions*, pages 8–12, May 2004.
13. J. Sneyers. *Optimizing Compilation and Computational Complexity of Constraint Handling Rules.* PhD thesis, K.U.Leuven, Belgium, Nov. 2008.
14. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS*, 31(2), Feb. 2009.
15. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *TPLP*, 10(1):1–47, 2010.
16. M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.
17. A. Swami and A. Gupta. Optimization of large join queries. *SIGMOD Rec.*, 17(3):8–17, 1988.
18. A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 345–354, Washington, DC, USA, 1993. IEEE Computer Society.
19. P. Van Weert. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, 2010. To appear.
20. P. Van Weert, L. De Koninck, and J. Sneyers. A proposal for a next generation of CHR. In *CHR '09*, pages 77–93, July 2009.
21. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *CHR '05*, pages 47–62, 2005.
22. P. Van Weert, P. Wuille, T. Schrijvers, and B. Demoen. CHR for imperative host languages. volume 5388 of *LNAI*, pages 161–212. Springer, Dec. 2008.
23. P. Wuille, T. Schrijvers, and B. Demoen. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [4], pages 123–137.